Raissa Nunes Peret (2024.1.08.021) e Ana Flávia Freiria Rodrigues (2024.1.08.001)

Análise e Desenho - Lista de Exercícios de Engenharia de Software

---

1) **O que é arquitetura de software e por que ela é importante no desenvolvimento de sistemas?**

   A arquitetura de software é como o "esqueleto" de um sistema, ou seja, a estrutura principal que define como as partes do software vão se organizar e se comunicar. É como planejar a planta de uma casa: você precisa definir quantos cômodos vão ter, onde cada coisa vai ficar, como será a distribuição de energia e água, para que no final a casa funcione bem.

   No desenvolvimento de sistemas, a arquitetura é super importante porque garante que o software seja bem organizado, escalável (ou seja, possa crescer conforme as necessidades), fácil de manter e seguro. Uma arquitetura bem pensada ajuda a evitar futuros problemas que seriam difíceis de corrigir quando o sistema já estiver em uso.

---

2) **Concorda ou discorda da frase: “As atividades de desenho estão diretamente ligadas aos requisitos não funcionais do sistema”?**

   Concordo! As atividades de "desenho" (ou design) de um software são diretamente influenciadas pelos **requisitos não funcionais**, que são aspectos como desempenho, segurança, escalabilidade e usabilidade. Por exemplo, se o sistema precisa ser super rápido, o design precisa ser pensado para garantir essa rapidez, escolhendo bem como os dados serão tratados ou como as funcionalidades vão ser integradas.

   Ou seja, quando você vai desenhar um sistema, você não foca só no que o software precisa **fazer** (os requisitos funcionais), mas também **como** ele precisa se comportar para garantir a qualidade e a experiência desejadas (os requisitos não funcionais).

---

3) **Quais os principais aspectos a serem considerados na criação da arquitetura/modelo de design de software?**

   Ao criar a arquitetura ou o modelo de design de um software, é importante considerar alguns pontos:

- **Escalabilidade**: O software vai conseguir crescer ou suportar um aumento no número de usuários sem perder qualidade?

- **Segurança**: Existem pontos na arquitetura que podem expor dados sensíveis ou tornar o sistema vulnerável a ataques?
- **Desempenho**: O sistema vai ser rápido o suficiente? Como os dados e os processos serão organizados para garantir que tudo funcione bem e rapidamente?
- **Manutenibilidade**: Será fácil corrigir erros, atualizar e adicionar novas funcionalidades no futuro?
- **Modularidade**: As partes do sistema estão organizadas de forma que possam ser trabalhadas separadamente, facilitando mudanças em uma parte sem afetar todo o sistema?

Esses são aspectos que ajudam a definir uma boa arquitetura, pensando tanto no presente quanto no futuro do sistema.

---

4) O que são design patterns e qual a importância deles nas atividades de desenho?

Design patterns (ou padrões de design) são soluções "prontas" ou modelos testados e comprovados para resolver problemas recorrentes no desenvolvimento de software. Eles são como receitas de bolo: você já sabe que elas funcionam porque foram usadas várias vezes e trazem resultados confiáveis.

A importância dos design patterns é que eles ajudam a acelerar o processo de design, porque você não precisa inventar uma nova solução do zero toda vez que enfrenta um problema comum. Eles também ajudam a criar sistemas que são mais fáceis de entender e manter, já que os padrões são amplamente conhecidos na comunidade de desenvolvedores.

---

5) O que é coesão e acoplamento. Explique como esses conceitos estão relacionados com a definição da arquitetura de software.

- **Coesão** é o quanto as partes de um módulo ou componente de software estão relacionadas e trabalham juntas para realizar uma função específica. Ou seja, quanto mais coeso for um componente, mais focado ele estará em realizar uma única tarefa bem definida. Um exemplo seria uma função que só calcula impostos, e nada mais. Isso é uma alta coesão, porque ela tem uma única responsabilidade.
- **Acoplamento** é o grau de dependência entre diferentes módulos ou componentes. Quando dizemos que dois módulos estão fortemente acoplados, significa que eles dependem muito um do outro, e se você mudar um, o outro pode ser afetado. O ideal é ter **baixo acoplamento**, ou seja, que os módulos sejam mais independentes e possam funcionar com menos interferência entre si.

Na arquitetura de software, **a relação de alta coesão e baixo acoplamento com a arquitetura de software é o objetivo ideal**. Isso porque:

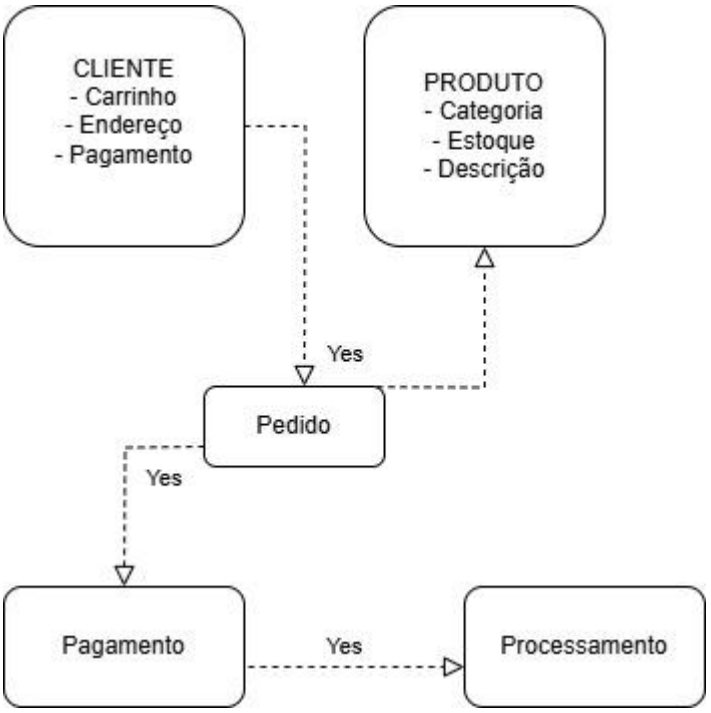
- **Alta coesão** garante que cada parte do sistema tenha uma função clara e bem definida, facilitando a manutenção, teste e evolução do software. Se cada módulo faz apenas uma coisa, fica mais fácil identificar e corrigir problemas ou fazer melhorias.
- **Baixo acoplamento** permite que as partes do sistema sejam alteradas, aprimoradas ou substituídas sem impactar muito o resto do software. Isso também melhora a escalabilidade, porque o sistema é mais modular e flexível.

  Se a arquitetura não considera esses conceitos, o software pode ficar confuso, difícil de manter e propenso a erros, porque mudanças em uma parte podem quebrar várias outras partes, especialmente se houver acoplamento forte e coesão baixa.

---

6) Dê exemplos de divisão lógica de sistema em pacotes. Obs. Os exemplos devem mostrar diagramas

1) ***Sistema de Gerenciamento de Biblioteca****: **Pacote "Cliente"**:*



Pacote “Cliente”:

- Contém classes relacionadas a clientes.
- Exemplos de classes:
  - Carrinho de compra: gerencia os itens que o cliente adiciona ao carrinho.
  - Endereço: formações de endereço de entrega do cliente.

    - Métodos de pagamento: classes para gerenciar os métodos de pagamento preferidos.

Pacote "Produto":

- Contém classes relacionadas a produtos.
- Exemplos de classes:
    - Categoria: define diferentes categorias de produtos.
    - Estoque: monitora a quantidade disponível de cada produto.
    - Descrição: descreve os atributos detalhados de cada item.

Pacote "Pedido":

- Gerencia o processo de pedidos.
- Exemplos de classes:
    - Ação de realização de pedido

Pacote "Pagamento":

- Contém classes para lidar com diferentes métodos de pagamento.
- Exemplos de classes:

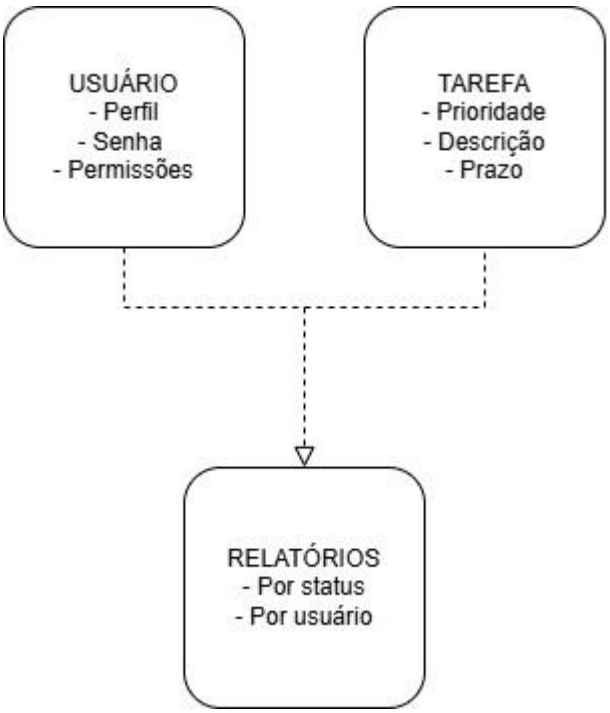
  - Cartão de crédito/débito, pagamento via Pix,

Pacote Processamento

- Realiza o processamento no sistema.

---

2) Sistema de Gerenciamento de Tarefas



Pacote “Usuário”:

- Contém classes relacionadas a clientes.
- Exemplos de classes:
    - Perfil: para guardar dados do usuário.
    - Senha: para segurança do perfil do usuário.
    - Permissões: permissões requisitadas para o acompanhamento do usuário no sistema.

Pacote “Tarefas”:

- Contém classes relacionadas às tarefas.
- Exemplos de classes:
    - Prioridade: nível de prioridade em que cada tarefa deve ser realizada, tarefas com maior prioridade devem ser realizadas primeiro.
    - Descrição: informações sobre a tarefa.
    - Prazo: tempo necessário para realização da tarefa.

Pacote “Relatório”:

- Contém classes relacionados aos relatórios..
- Exemplos de classes:
    - Por status: relatório de acordo com processo.
    - Usuário: relatório de acordo com as informações do usuário.

---

7) Qual a importância da disciplina de desenho para a experiência do usuário?

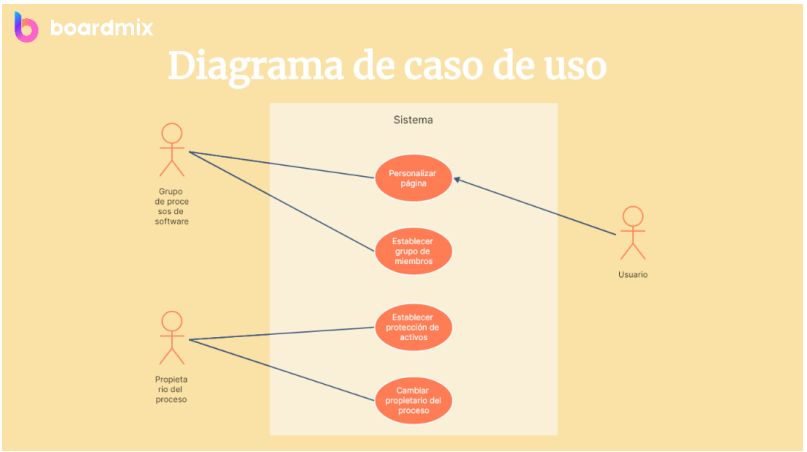
O desenho tem um papel crucial na experiência do usuário porque é onde as ideias abstratas começam a ganhar forma visual e funcional. Ele permite que designers criem protótipos e esboços para explorar como os usuários irão interagir com um produto. Além disso, o desenho ajuda a prever problemas de usabilidade e a otimizar o fluxo de navegação. Por exemplo, ao desenhar interfaces de sites ou aplicativos, é possível testar diferentes layouts, tamanhos de botões e disposição dos elementos para garantir que a interação seja simples e intuitiva, o que resulta em uma experiência mais satisfatória para o usuário.

---

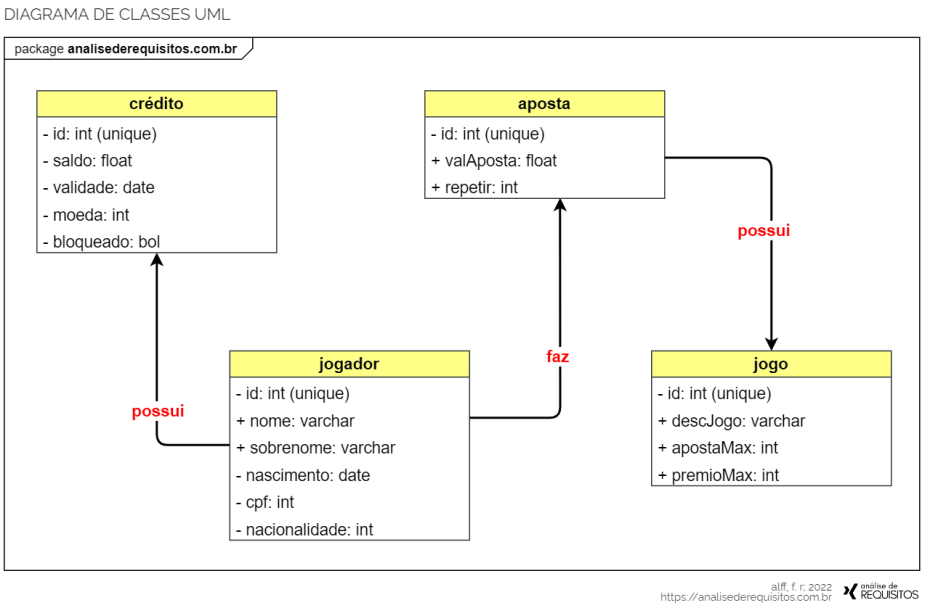
8) Como a UML é utilizada nas atividades de desenho? Dê exemplos de diagramas.

O desenho tem um papel crucial na experiência do usuário porque é onde as ideias abstratas começam a ganhar forma visual e funcional. Ele permite que designers criem protótipos e esboços para explorar como os usuários irão interagir com um produto. Além disso, o desenho ajuda a prever problemas de usabilidade e a otimizar o fluxo de navegação. Por exemplo, ao desenhar interfaces de sites ou aplicativos, é possível testar diferentes layouts, tamanhos de botões e disposição dos elementos para garantir que a interação seja simples e intuitiva, o que resulta em uma experiência mais satisfatória para o usuário.

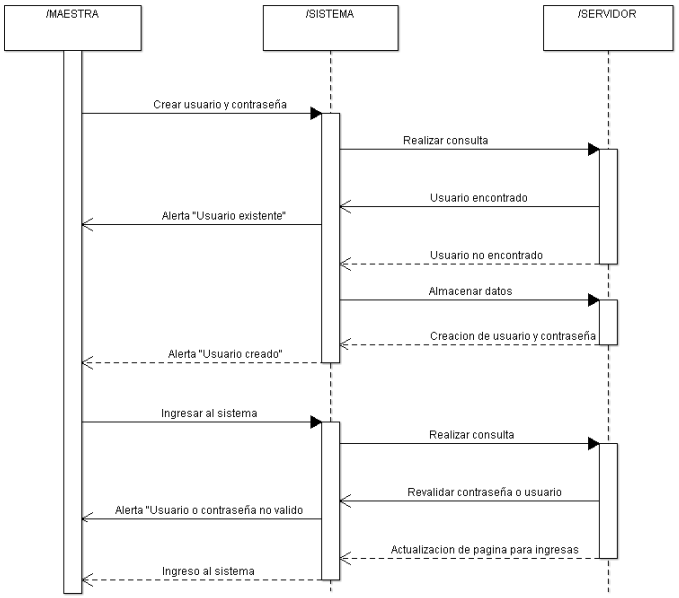
- UML nas atividades de desenho
  A UML (Unified Modeling Language) é uma ferramenta importante para documentar e planejar o desenvolvimento de sistemas complexos. Ela ajuda a visualizar a estrutura e o comportamento de um software antes mesmo de ser codificado. Alguns exemplos de diagramas que são amplamente utilizados no processo de desenho de software são:
- *Diagrama de casos de uso*: define as interações entre usuários e o sistema, ajudando a identificar requisitos funcionais.



- *Diagrama de classes*: mostra a estrutura de um sistema, incluindo classes, atributos e relações entre elas, essencial para o design da arquitetura.



- *Diagrama de sequência*: detalha a ordem dos eventos que ocorrem durante a execução de uma funcionalidade, focando na comunicação entre objetos.



Esses diagramas ajudam a organizar as ideias e a garantir que todos os envolvidos no projeto tenham uma visão clara de como o sistema vai funcionar.

9) Como se aborda a integração de sistemas legados no desenho de um novo software?

A integração de sistemas legados em um novo software é um desafio, mas pode ser feita de forma eficiente com um planejamento adequado no desenho do sistema. Primeiramente, é importante mapear as funcionalidades críticas do sistema antigo e garantir que o novo sistema seja compatível com elas. Isso pode envolver o uso de APIs, adaptadores ou até a reescrita parcial do código legado. Além disso, é necessário avaliar os dados existentes para garantir que eles sejam transferidos ou acessados corretamente no novo sistema. O objetivo principal é garantir que a transição seja suave, sem perda de funcionalidades essenciais ou dados, enquanto se moderniza a tecnologia usada.

Outubro, 2024.